A Brief Introduction to Python Scripting

Python is documented down to the last detail at the Python language website http://python.org and the SourceForge website http://python.sourceforge.net. There are now many books discussing the use of Python and how to use it with other open source software. We can recommend the free tutorial available at www.python.org in the documentation section. The book "Learning Python" from O'Reilly Press is another excellent resource. This section provides a very brief introduction to some basic concepts aimed at "getting started". The interested reader can refer to any of the other sources referred to above for details.

We begin with the traditional "Hello World" program. To start the python interpreter type `python` at the command prompt:

```
% python
Python 2.2.1c2 (#1, Apr 11 2002, 12:36:10)
[GCC egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
```

Programs can be placed in files with a .py extension. The `helloworld.py` program would contain the following lines:

```
# helloworld.py
print "Hello World"
```

The program can then be executed by providing the name of the program file to the interpreter as follows:

```
% python helloworld.py
Hello World
%
```

Variables and arithmetic expressions

A simple program can best illustrate the use of arithmetic expressions.

```
# simple_program.py
pi = 3.1415
degs = 0
while degs <= 90:
    rads = degs * pi / 180.0
    print degs, " degrees = ", rads, " radians"
    degs = degs + 10.
```

The output of this program is the following table:

```
0  degrees =  0.0  radians
10.0  degrees =  0.174527777778  radians
20.0  degrees =  0.349055555556  radians
30.0  degrees =  0.523583333333  radians
40.0  degrees =  0.698111111111  radians
50.0  degrees =  0.872638888889  radians
60.0  degrees =  1.04716666667  radians
70.0  degrees =  1.22169444444  radians
80.0  degrees =  1.39622222222  radians
```

```
90.0  degrees =  1.57075  radians
```

Python is a dynamically typed language (i.e you do not need to declare the variable types apriori) and the names can represent different types depending on the arithmetic operations performed. In the above example, "degs" was initially set to the integer 0, and is seen in the first line printed. Subsequently a real number (10.) is added at each stage of the loop and this changed the type of "degs" as it can be seen that "degs" values printed are real. Note that the above example is contrived to illustrate this point and there is no real reason to add a real value 10.0 in the loop.

The output of the program looks less than ideally formatted. To make it look better, we can make use of format strings. For example:

```
>>> print "%3d degrees = %0.4f radians" %(degs, rads)
```

would produce output that looks like this:

```
 0 degrees = 0.0000 radians
10 degrees = 0.1745 radians
20 degrees = 0.3491 radians
30 degrees = 0.5236 radians
40 degrees = 0.6981 radians
50 degrees = 0.8726 radians
60 degrees = 1.0472 radians
70 degrees = 1.2217 radians
80 degrees = 1.3962 radians
90 degrees = 1.5708 radians
```

The format strings ***%d***, ***%s*** denote integers and strings respectively, and ***%g*** and ***%f*** denote floats.

Conditional statements

The ***if*** and ***else*** statements provide an easy way to perform tests. For instance:

```
if x != y:
    print 'x is not equal to y'
else:
    print 'x and y are equal'
```

The indentation is required to isolate the ***if*** and ***else*** clauses, but the else clause is optional. Do nothing clauses can be created by using the ***pass*** statement.

```
if x != y:
    pass
else:
    print 'x and y are equal!'
```

Multiple test cases can be implemented using the ***elif*** clause.

```
if x == 'n':
    print 'Answered no'
elif x == 'y':
    print 'Answered yes'
else:
    print 'invalid answer'
```

Boolean expressions can be formed by using ***or***, ***and***, and ***not*** keywords.

```
if x > y and z > x:
    print 'z is the max value'
if not (x==z or y==z or x==y):
    print 'There are no equal values'
```

File Input and Output

To open a text file and read its contents you would:

```
# Get a file object
f = open("myfile.txt")

# The readline() method is invoked on file
# and one line is read from the file.
line = f.readline()

# The following section keeps printing and reading
# subsequent lines of data while there are new lines to
# be read
while line:
    print line
    line = f.readline()

# The while loop is exited when there are no more lines
# to be read. To close the open file:
f.close()
```

To write output to a file:

```
fout = open("out.txt", 'w')
# The `w' indicates file should be opened for writing
# and is created if it does not already exist.

fout.write("hello\n")
# or equivalently
print >> fout, "hello"

fout.close()
```

Lists and Tuples

Lists and tuples are sequences of arbitrary objects. Lists can be created by:

```
mylist = ["a", 1.0, "c", 4]
```

Note that you can mix items of any type in a list. You can also have lists nested inside lists.

```
my_other_list = ["a", "b", [1,2,3], "d"]
```

The lists are indexed by integers starting with zero. To see the first item in mylist, you would:

```
firstitem = mylist[0]         # returns the item "a"
```

```
# To set an item:
mylist[2] = "x"               # Changes the item in the index position 2.
print mylist
# Output: ["a", 1.0, "x", 4]
```

To append new members to the list, the ***append()*** method is used as follows:

```
mylist.append("nextitem")
print mylist
# Output: ["a", 1.0, "x", 4, "nextitem"]
```

Similarly, individual items can be removed from the list using the ***remove()*** method:

```
mylist.remove(1.0)
print mylist
# Output: ["a", "x", 4, "nextitem"]
```

You can also insert items into specific positions:

```
mylist.insert(1, "inserted_item")
print mylist
# Output: ["a", "inserted_item", "x", 4, "nextitem"]
```

Lists can be concatenated by using the ***"+" operator***:

```
newlist = mylist + [9, 10, 11]
print newlist
# Output: ["a", "inserted_item", "x", 4, "nextitem", 9, 10, 11]
```

Tuples are very similar to lists and are constructed by using parentheses instead of brackets or just a comma−separated list.

```
mytuple = (1, 2, -3)
myvector = (magnitude, direction)
# is the same as
myvector = magnitude, direction
```

Tuples support the same operations as are supported by lists except the appending or modification of elements after they are created. That is, you cannot modify elements or append elements to a tuple.

Loops

We saw the simple loop using the ***while*** statement in the previous examples. Other looping constructs such as the ***for*** statement are available to the user. It is important to note that statements within the loop are indented. An example of its usage is:

```
for i in range(3):
    print "10 raised to ", i, " is ", 10**i
```

Will produce the result:

```
10 raised to 0 is 1
10 raised to 1 is 10
10 raised to 2 is 100
```

Note that *a=range(3)* is equivalent to *a=[0,1,2]***.** Similarly the *range(1, 6)* is equivalent to *[1,2,3,4,5]* and *range(10, 8, −1)* is equivalent to *[10, 9]*. To generalize, ***range(i, j, k)*** produces a list of integers from i to j−1 using a stride k. If i is omitted, it is taken to be zero and k defaults to 1 if omitted. A more efficient (in terms of memory and runtime) is the ***xrange()*** function used exactly like *range()*.

One can also loop through lists using the for statement so:

```
mylist = ['a', 'b', 3]
for item in mylist:
    print item
```

Produces the output:

```
a
b
3
```

Dictionaries

A dictionary allows you to associate values index by keys. Dictionaries can be created using values in curly braces like this:

```
a = {
      "name" : "CCM3",
      "center" : "NCAR,"
      "model_of" : "Atmosphere"
     }
```

To access members of the dictionary, we use the key−indexing facility:

```
model_name = a["model"]
modelling_center = a["center"]
```

The ***keys*** associated with a dictionary can be obtained as a list by:

```
b = a.keys()
```

You can check the dictionary membership by using the ***has_key()*** method:

```
if a.has_key("model_of"):
    print a["model_of"]
else:
    print "Unknown model of"
```

Functions

A function can be created using the ***def*** statement as shown below.

```
def myadd(a, b):
    c = a*10 + b
```

```
    return c
```

Note once again that the statements inside the function are indented after the "*def*" statement. To invoke the function we do the following:

```
addvalue = myadd(2, 5)
```

It is possible to return multiple values using comma separated names in the return statement inside the function.

```
def myaddsub(a, b):
    c = a*10 + b
    d = a*10 - b
    return c, d
```

In this case the function is invoked as follows:

```
addvalue, subvalue = myaddsub(2, 5)
```

The function definition can also be done in a way such that **default values** for input parameters can be set.

```
def myaddsub(a, b, base=10):
    c = a*base + b
    d = a*base - b
    return c, d
```

Then, when you need base to take a different value than the default 10, you can:

```
addvalue, subvalue = myaddsub(2, 5, base=2)
```

Modules

To keep your programs manageable as they grow in size, you may want to break them up into several files. Python allows you to put multiple function definitions into a file and use them as a module that can be imported into other scripts and programs. These files must have a .py extension. For example:

```
# file my_function.py

def minmax(a,b):
    if a <= b:
        min, max = a, b
    else:
        min, max = b, a

    return min, max
```

To use the above module in other programs, you would use the import statement.

```
import my_function

x,y = my_function.minmax(25, 6.3)
```